Seven (+-2) Sins of Concurrency

Chen Shapira
HP Software-as-a-Service
cshapi@gmail.com

Databases are normally high-concurrency environments, where the number of concurrent processes ranges from tens to thousands. Yet many developers are either ignorant of the complexities and traps that are inherent in multi-process systems or wrongly assume that all the problems will be correctly handled by the database. There are many ways in which code that ran correctly on the developer's workstation will slow down, crash, give wrong results or otherwise fail to scale when it is deployed on production environment.

In this paper, I'll present seven common concurrency mistakes well-known in computer science theory, yet often ignored by database developers. I'll show test-cases that reproduce each problem within Oracle and give tips on how to identify the problem when it occurs in production environment.

Introduction

At 1967, the first computer systems to support multiprogramming were released to the market [1]. Until that time, computers could only run one program at a time, sequentially. If your program was busy reading IO, or a user spent some time thinking instead of using the program, expensive CPU time was wasted.

Multiprogramming allows the operating system to switch between processes, so while one program is waiting for the user or an IO operation, other programs can use the CPU. Modern operating systems can do this context switching so fast and seamlessly that even on a single CPU machine; we can run multiple users or multiple programs and believe that they all run at the same time.

Multiprogramming systems make better use of their CPU and give users better response times. Without doubt, this is a very good thing. There is a catch, of course. Programmers can no longer assume their code will run continually and without interruptions on the CPU. If your code writes to a file, it may lose the CPU in the middle of writing and another program will run, possibly deleting this file. Programmers need to learn to write code that will expect these situations and can handle them correctly.

Even though multiprogramming has been around since 1967, there is some evidence that not every program running today is designed to correctly handle issues that arise when several programs are running concurrently.

Due to the complexity of concurrency issues, they are normally studied using sample programs, which recreate specific concurrency problems without the added complexity associated with real systems.

In this paper, I start by discussing the most complicated concurrency problems – the race condition. I give multiple examples and show the trade-offs involved in resolving it. Then I discuss some of the classical problems of concurrency, problems that were shown in 1970s by operating system researches. I show how these problems appear in Oracle and show some of the conclusions that can be reached by studying them.
I finish the paper with quick review of concurrency problems that are unique to Oracle and related to its read consistency mechanisms.

Sin 1: Race Condition

Assume we have a bank account that is shared by multiple users. Users will want to deposit and withdraw money from the account.

We can create the account and interface with the following code:

```
create table bank_account (id number primary key, amount
number);
insert into bank_account values (1,0);
commit;


create or replace procedure update_account(p_id
number,p_amount number) as    n number;
begin
     select amount into n from bank_account where id=p_id;
     update bank_account set amount = n+p_amount
     where id=p_id;
end;
/


create or replace procedure deposit (p_id number,p_amount
number) as
begin
     update_account(p_id,p_amount);
end;
/

create or replace procedure withdraw(p_id number,p_amount
number) as
begin
     update_account(p_id,p_amount*(-1));
end;
```

And now two users attempt to use the bank account concurrently:

```
SQL> exec deposit(1,500)
                                   SQL> exec withdraw(1,-500)

SQL> commit;

                                   SQL> commit;
```

```
SQL> select amount from bank_account;
AMOUNT
----------
-500
```

As you can see from this simple example – a code that was not written to support concurrent users ends up giving a wrong result when used by two users.
When the result of a procedure depends on the order in which different users executed it, we call this a **race condition**. The example we just saw shows a specific type of race condition, known as a **lost update.** Race conditions are probably the most annoying bugs to find and solve, as most of the time everything works perfectly fine, and only on rare conditions something unexpected happens. In particular, it is important to keep in mind that race conditions are sensitive to exact timing of operations and environments that change the timing such as a debugger or trace may prevent a reproduction of the bug . For this reason, it is important to instrument the program to keep log of important events and their timing while running. Often this log is the only way to track down the bug leading to a race condition when the problem would not reproduce.

Since race conditions are rare and difficult to reproduce, it is tempting to simply ignore them. This method is known as **Ostrich Algorithm**, which I consider **Sin 2**. While it is sometimes a correct business decision not to spend resources fixing a very rare issue, it is important to remember that a bug which occurs very rarely in QA with 5 testers working on the code, will occur with great frequency on a busy production environment with 5000 users working on the system.

The following code fragment is a demonstration of how common is the Ostrich algorithm when dealing with potential race conditions:

```
spool XXX_drop_db_links.sql

select 'drop database link '||OBJECT_NAME||';'
from obj
where OBJECT_TYPE='DATABASE LINK';

spool off

@XXX_drop_db_links.sql
```

This code fragment should be very familiar to most readers. In order to save time which executing the same DDL multiple times, we write a query to generate the DDL, spool the results into a file, and then execute the contents of the file.
It should be clear that in case of two users executing this code, the results will be unpredictable. Despite the clear potential problems, this pattern is quite commonly used in production systems - demonstrating the usefulness of the Ostrich algorithm in resolving concurrency problems.

How do we avoid race conditions? The key to preventing race conditions would be to identify resources which should not be modified by two processes at the same time, and prevent this from occurring. In other words, we need **mutual exclusion** –a way of making sure that while one session is accessing a shared resource, others will be excluded from it. To put it in more useful terms, we will refer to the section of code in which exclusive access to the shared resource is needed as a **critical section** and instead of seeking to protect the shared resource, we will have the easier task of making sure no two processes are running the critical section of the code at the same time. Note that during normal operations, a transaction in Oracle defines a critical section – locks are held from the first modification to the point the transaction is closed by either commit or rollback.

Although this requirement is enough to avoid race conditions, it is not sufficient to ensure efficient concurrency. A good solution will need to follow these conditions [2]:

1. No two processes may be simultaneously inside their critical sections
2. No assumptions may be made about the number or speed of CPUs in the system
3. No process running outside the critical section may block other processes
4. No process should have to wait forever to enter its critical section

Oracle has three mechanisms for protecting shared memory and providing mutual exclusion. These are the **locks, latches** and **mutexes**.

Oracle documentation defines **locks** [3]:

> “Locks are mechanisms that prevent destructive interaction between transactions accessing the same resource—either user objects such as tables and rows or system objects not visible to users, such as shared data structures in memory and data dictionary rows.”

In this paper I'll mostly look at row locks, taken either automatically when a row is modified by DML, or by user request using `select … for update` syntax. However, these are far from being the only lock types used by Oracle.

Latches are often known as "lightweight locks", in this context lightweight means that they take fewer bytes in memory and are requested and released faster. They are typically used to protect internal memory structures such as blocks in the buffer cache or cursors in the shared pool. Latches are designed to be held for extremely short amounts of time. **Mutexes** are very similar to latches, but even lighter (see a trend here?). Since version 10g, Oracle has been replacing many of the latches by mutexes. Developers who are used to POSIX mutexes should remember that these have nothing in common with Oracle mutexes, which are implemented by Oracle and not by your standard library.

Oracle also gives an interface that allows users to request a lock and give it a unique name, without having the lock tied to any particular database object. This can be used to protect objects that are not normally protected by Oracle lock services, such as writes to a file. It can be used to serialize procedures, in case we want certain actions to run in a specific order or at the exact same time [4]. User locks can also be used in a RAC environment in cases only one session in each node is allowed to run a specific action (perhaps one that is tied to the specific physical machine).

Here is an example of how user locks can be used to protect code that writes to an external file from being executed simultaneously. Start by wrapping the DBMS_LOCK code with two useful functions – one that creates a lock names "synchronize" and the other which releases it:

```
create or replace procedure lock_sync
  as
    n1       number(38);
    m_handle varchar2(60);
  begin
    dbms_lock.allocate_unique
    ( lockname => 'Synchronize'
    , lockhandle => m_handle
    );
    n1 := dbms_lock.request
    ( lockhandle => m_handle
    , lockmode => dbms_lock.x_mode
    , timeout => dbms_lock.maxwait
    , release_on_commit => false - default!
    );
  end lock_sync;
  /
```

```
create or replace procedure release_sync
  as
    n1       number;
    m_handle varchar2(60);
  begin
    dbms_lock.allocate_unique('Synchronize',m_handle);
    n1 := dbms_lock.release(m_handle);
  end release_sync;
/
```

Now we can use our custom locks to make sure that sessions will execute the critical section one at a time:

```
exec lock_sync

spool XXX_drop_db_links.sql

select 'drop database link '||OBJECT_NAME||';'
from obj
where OBJECT_TYPE='DATABASE LINK';

spool off

@XXX_drop_db_links.sql



exec release_sync
```

Now the first user will acquire the lock, and will hold it until he finishes running the critical section. Then he will release the lock, allowing the next use to acquire it. If the second user tried to acquire the lock before the first one finished, he would have had to wait until the first user released the lock. This way, the second use writing to the file will not interfere with the first user executing the contents of the file and the results will be predictable.

Note: the last parameter of dbms_lock.request is release_on_commit. Releasing locks on commit is the default behavior of Oracle’s automatic locks, but not of user locks. Calling dbms_lock.request without this parameter will create a lock that will not be released on commit. This is sometimes beneficial, in our example, the critical section executes DDLs and we do not want their implicit commits to release the lock, but it is important to be aware of this.

Sometimes, solving race conditions can lead to different kinds of issues. Take a look at another race condition:

```
select max(id) into max_id from my_table;

insert into my_table values (max_id+1,some_data);

commit;
```

I wish I could have said that this is a theoretical example that never happens in production.
It is quite clear in this example that the developer wanted to generate a unique ID for his table, and either never heard of sequences or greatly disliked them.
This code contains a race condition that can lead to duplicate IDs inserted into the table.

It may be tempting to protect the maximum id in the table using `select for update`, however there is no way to actually do this.

Our first attempt:

```
SQL> select max(id) into max_id from my_table for update;
select max(id) into max_id from my_table for update
                                         *
ERROR at line 1:
ORA-01786: FOR UPDATE of this query expression is not
allowed
```

Fails. `For update` is only allowed on actual specific rows, not on aggregate expressions.

We can try to trick SQL into locking the maximum id:

```
select id into max_id from my_table where id=(select max(id) from my_table) for
update;

insert into my_table values (max_id+1,some_data);

commit;
```

This attempt fails in a more subtle and therefore more destructive way. Assume that currently the maximum ID in our table is 100. The session 1 selects 100 as max_id and inserts id 101, which is now the new maximum.

Suppose that at this point in time, session 2 runs. It will block on select for update and wait until session 1 commits. Once session 1 commits, it will retrieve 100 into max_id, since this is the row it waited on! It will not get the new maximum id, although now that session 1 committed, the change is visible to session 2. However, since session 2 waited for row 100, it will get 100 as max_id, and insert a duplicate maximum ID.

This code snippet is actually worse than the original, we are wasting time waiting for the wrong row, and then still have the duplicate id bug.

We can make another attempt to solve the race condition. Since we cannot lock the maximum id in our table, we can try to keep it in another table:

```
select max_id into p_max_id from extra_table for update;

insert into my_table values (max_id+1,some_data);

update extra_table set max_id=max_id+1;

commit;
```

This solution will not allow duplicate inserts. However, Oracle used to support concurrent inserts. Under this solution, only one session can insert data to my_table at the same time. The other tables will wait. By solving the race condition, we created **Sin 3 - serialization**. As we saw before, we can also create serialization without solving any race conditions (Tom Kyte says that “Blocked update or delete indicates that you probably have a lost update problem in your code” [5]).
Serialization is not as serious bug as a race condition. While it severely hurts performance, it does not always lead to incorrect results. Since protecting critical sections against race conditions leads to serialization, it is important to be careful to protect only the parts that must be protected, and make sure that the serialization is 100% necessary. In the example above, sequences can be use successfully to generate new IDs without creating serialization. In case where sequences cannot be used (non-oracle database perhaps?), it is still possible to avoid serialization. For example by dividing future ids between the servers that perform the inserts – one server can insert only even numbers and the other only odd numbers (This solution can be generalized to any number of servers by using mod() function. Implementation details of this solution are left as an exercise to the reader).

Operating systems literature is full of interesting concurrency problems that have been discussed and analyzed. It is useful to learn to recognize and resolve concurrency problems using these classical problems, rather than using real production code that is much more complex.

Dining Philosophers

Five philosophers are sitting around a dining table. Each philosopher has a plate with rice, which requires a pair of chopsticks to eat. Between each two philosophers is a single chopstick.

Philosophers never speak to each other. They either eat or think. When a philosopher gets hungry, he attempts to acquire his chopsticks, one at a time. Then he eats for a while, and after eating he puts his chopsticks down and resumes thinking. Is it possible to write a program for each philosopher that does what it is supposed to do and never get stuck?

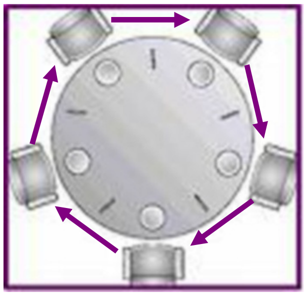
The obvious solution is for each philosopher to run the following code:

```
-- number of philosophers

select count(*) into N from sticks;

think();

-- take right chopstick
update sticks set owner=philosopher_id where s_id=p_id;

-- take left chopstick
update sticks set owner=philosopher_id where
s_id=mod(p_id+1,N);

eat(); -- nom nom nom

commit; -- put down chopsticks
```

Imagine, however, a situation where all philosophers pick up their right chopstick simultaneously. None of them will be able to take their left chopstick, since each philosopher will wait for its neighbor. The result is a **Sin 4- deadlock**.

Oracle will recognize this situation as a deadlock. At this point Oracle will select one of the philosophers and rollback his last statement, the attempt to pick up the left chopstick. Now this philosopher is no longer blocked and can proceed. How he proceeds is up to us, but we should note that at this point the philosopher is still holding his right chopstick and therefore all the other philosophers are still blocked. The other philosophers will not eat until he releases the right chopstick, either by rolling back the transaction or committing. The demo code re-raises the error after logging it, which causes the process to exit – killing the unlucky philosopher. Once this philosopher is dead, the neighbor on his right has a guaranteed access to his left chopstick, preventing additional deadlocks.

In the diagram below you can see the cycle of processes, each waiting for the next one to finish running and release resources. This cycle is the defining characteristic of deadlocks - deadlock detection algorithms work by recognizing cycles of blocked processes. Deadlocks are the easiest problem since Oracle finds it for you (compare with race conditions that are notoriously difficult to find). Mark J Bobak has very good paper about finding the problems that lead to deadlocks [6].



We could modify the program so that after picking up the right chopstick, the program checks to see if the left chopstick is available. If it is not, it will put down the right chopstick, think a bit more and then retry.

```
think();

-- pick up right chopstick
update sticks set owner=in_p_id where s_id=in_p_id;

-- check if left chopstick is available
select s_id into r_s from sticks where
s_id=mod(in_p_id+1,N) for update nowait;

-- if it is, pick it up
update sticks set owner=in_p_id where
s_id=mod(in_p_id+1,N);

eat(); -- nom nom nom
commit; -- put chopsticks down
```

```
-- if left chopstick not available
exception
     when resource_busy then
          rollback; -- put down the right chopstick
```

This clearly prevents deadlocks from occurring, but there is a new problem. In jurisdictions where Murphy's Law applies, this can lead to a situation where all the philosophers start together, pick up the right chopstick, see that the left is unavailable, put it down, wait, pick up the right chopstick simultaneously again, and so on, forever. This is leads to **starvation – Sin 5.** All sessions continue working but fail to make any progress (The situation is also called a livelock, but livelock is used to describe several other problems as well, so it can be ambiguous).

Of course, under normal circumstances we would not expect complete starvation to occur. When the time our sessions spend eating or thinking is random and the sessions run for long periods, the probability for starvation is negligible. What we should be checking is how this method affects the time each philosopher has to wait before eating – the response times.

The "dining philosophers" code is instrumented, and each philosopher records in a log table how long he waited before eating. Below you can see statistics regarding the response times experienced by five philosophers, using the eating algorithm we just described.

```
PHILOSOPHER_ID   MAX_WAIT   AVG_WAIT TOTAL_WAIT   PCT_WAIT STDDDEV_WAIT
-------------- ---------- ---------- ---------- ---------- ------------
             1       .672  .23136413     42.571 70.9516667   .132490119
             2        .86 .237406593     43.208 72.0133333   .152001525
             4       1.25 .254491228     43.518      72.53   .184494552
             3        .86 .205541237     39.875 66.4583333    .14581332
             0       .797 .217721311     39.843     66.405   .140907631
```

Note that:

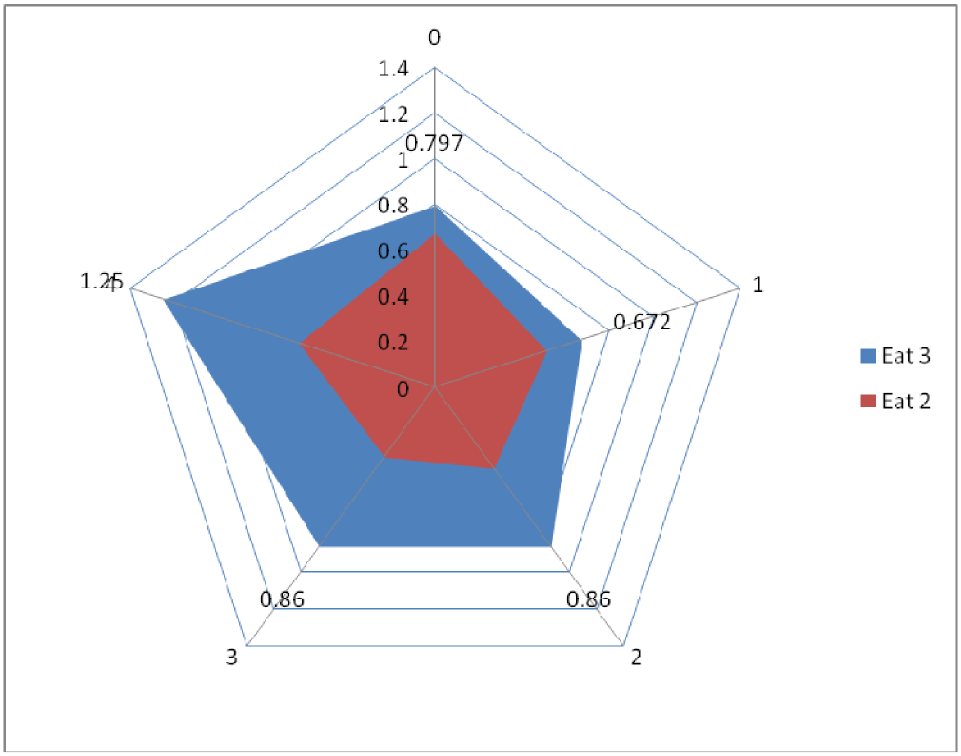
- There is significant variance between maximum wait times of different philosophers. Some philosophers would starve while others can be quite satisfied. We can also see the high variance in the stddev column.
- The maximum time spend waiting is quite high. Over a second in one case.
- Under ideal conditions, 2 out of 5 philosophers can eat simultaneously. So we would expect the percent of time spent waiting to be around 60%. We can see in the table above that 3 of the philosophers waited over 70% of the time. This indicates a waste of resources.

Compare these results with the response times when using a more optimal algorithm that will be shown later:

```
PHILOSOPHER_ID   MAX_WAIT   AVG_WAIT TOTAL_WAIT   PCT_WAIT STDDDEV_WAIT
-------------- ---------- ---------- ---------- ---------- ------------
             1       .515 .206848958     39.715 66.1916667   .087391237
             2       .438  .18968932     39.076 65.1266667   .082672322
             4       .625 .230628415     42.205 70.3416667   .112258158
             3       .375 .169275109     38.764 64.6066667    .07941457
             0       .672  .23995858     40.553 67.5883333    .11180214
```

You can immediately see that under a better algorithm, there is less time wasted on needless waits and also less variance between the results and the maximum time spent waiting is significantly lower.

You can see the different between maximum wait times in these algorithms clearly in the following graph:

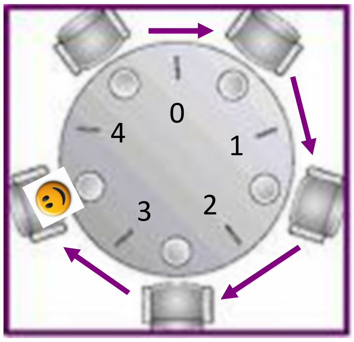


The blue pentagon represents the second algorithm, and it should be clear from the graph that it is much more wasteful of resources than the algorithm I will now introduce (represented by the red pentagon).

Let us number the chopsticks, and ask each philosopher to first attempt to pick up the lower numbered chopstick (instead of always picking up the right one first).

```
think();

update sticks set owner=in_p_id where
s_id=least(in_p_id,mod(in_p_id+1,N));

update sticks set owner=in_p_id where
s_id=greatest(in_p_id,mod(in_p_id+1,N));

eat();

commit;
```

This forces one of the philosophers to wait for his left chopstick – before he picks up his right chopstick! This frees the philosopher on his left to pick up both chopsticks and eat. Using this algorithm our philosophers don't need to keep retrying to pick up chopsticks while starving, but changing the order in which on philosopher waits for his chopsticks breaks the cycle and prevents deadlocks, as we can see below.



The "dining philosophers" problem models a **scalable** system. When you add more philosophers to the table, they arrive with at least part of the resources they will require to do their work. In a system with N philosophers, the number of philosophers eating is `floor(N/2)`, so as we add more philosophers, our system will approach 50% utilization. 50% utilization indicates **linear scalability** – the holy grail of scalable systems. So, while we do need to carefully avoid deadlocks and starvation, if our system resembles the "dining philosophers" problem, we are in a very fortunate situation.

Note about deadlocks and dining philosophers: Experience shows that the number one cause of deadlocks in Oracle is unindexed foreign keys. If you have a parent table and child table, where the foreign key on the child table is unindexed, deleting rows from the parent table will lock the entire child table, not just the affected rows. These unexpected extra locks seem to be the leading cause for deadlocks. If you encounter deadlocks in Oracle based programs, I suggest checking for foreign key indexes before you go looking for philosophers and chopsticks.

Barbershop Problem

A small town has one barbershop. Several barbers work in this barbershop. When someone in town decides it is time to get a haircut, he walks into the barbershop. If one of the barbers is free, the customer will sit down in the chair and receive a haircut. After the customer got a haircut, he will immediately leave. If all barbers are busy, the customer will sit in the waiting room until it is his turn to get a haircut.

When we model the barbershop, it will be important to make sure certain rules are observed:

- Only one barber will work on each customer
- A barber can only handle one customer at a time.
- Barbers will not attempt to cut hair of citizens who do not require a haircut.
- Barbers will accept customers on a first-come first-serve basis.
- A person who is waiting for a barber cannot enter the barbershop again until he got his haircut.

Our model contains two procedures and one table. The table contains a list of potential customers – people who at any given moment either wait for a barber or not.
The first procedure generates customers. It runs in a loop, and every second picks a random number of random people to enter the barbershop. The second procedure is executed by each barber – it looks at the table and selects the next person who requires a haircut, performs the haircut, marks that the person no longer needs his hair cut and continues to the next person.

Interesting implementation note: I implemented the barber code using the new `select for update skip locked` syntax, added in Oracle 11g. This allows the loop ran by each barber to open a cursor selecting all potential customers, but to only fetch rows that were not yet updated by other barbers and to only lock rows that were fetched. There are other possible implementations of barbers, but this seemed the most straightforward and efficient. It might not be a coincident that Oracle used `select for update skip locked` internally in its advanced queues long before it was officially supported as an interface.

The customers table looks like this:

```
create table customers
      (id number primary key,
      entered_shop timestamp, -- we want to track the wait times
      needs_cut number);

declare
begin
      for i in 1..2000 loop
            -- at the beginning not one needs a haircut
            insert into customers values (i,null,0);
      end loop;
      commit;
end;
/
```

The code for generating new customers:

```
loop
      update customers set needs_cut=1,entered_shop=systimestamp
      where id in (
            select id from
                  (select id from customers
                  where needs_cut=0
                  order by dbms_random.random)
            where
            rownum<=(dbms_random.value*(p_avg_customers_per_sec*2+1))
      );

      commit;
      dbms_lock.sleep(1);
end loop;
```

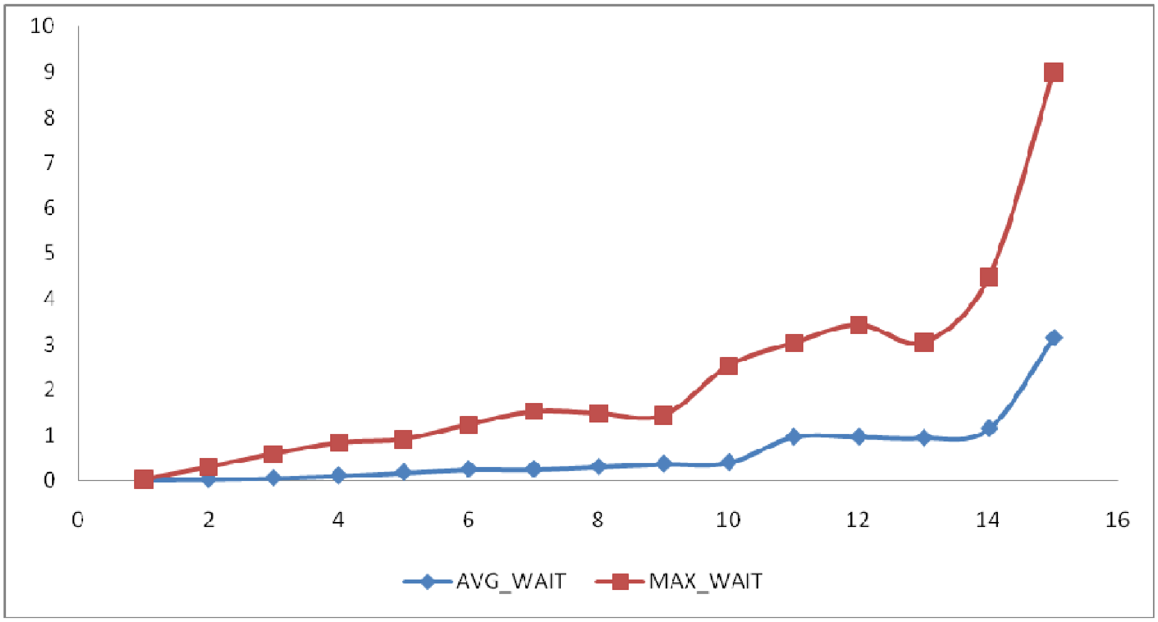
Each barber runs the following code:

```
cursor c is
      select * from customers where needs_cut=1
      order by entered_shop
      for update skip locked;

... - skipped boring parts here

loop
      open c;
      loop
            fetch c into l_rec;
            exit when c%NOTFOUND;
            cut_hair(dbms_random.value*p_avg_cut_time*2);
            finish_work(l_rec.id);
      end loop;
      commit;
      close c;
end loop;
```

This problem is interesting to study because it is a classical example of a **queue system**. The numbers of servers is fixed, customers need service at random intervals, and the length of service is also random. It is important to see that every concurrent system is also a queue system. When you have multiple processes running simultaneously on the same system, they will end up queuing for shared resources such as CPU, IO, specific blocks in memory, etc. Using the barbershop problem as a model, we can demonstrate concurrency mistakes that are queue related. Note that this brief overview is not intended to be a full queue theory course - this will be way beyond the scope of this paper. Cary

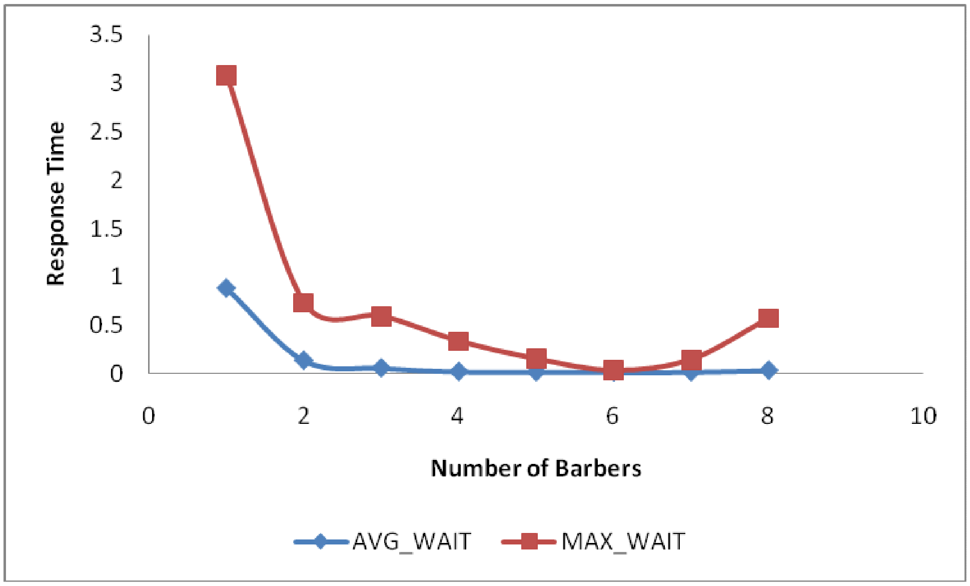
Millsap wrote an excellent review of queue theory and its application to Oracle [7] and Craig Shallahamer dedicated an entire book to this topic [8].

First, assume we have 3 barbers; each of them can finish a haircut in 0.3 seconds on average. How many customers per second can our shop handle?



One may expect that with 3 barbers, each capable of handling 3 customers per second, we would not see response time degradation before hitting the 9 customer limit. However, from the graph we can see that the wait times start getting longer significantly before we reach the theoretical limit of 9 customers per second, but after this point, the wait times increase sharply. Sizing the system on the verge of **over-utilization**, where tiny random effects can severely impact response times is **Sin 6**. We can also see that we did not quite achieve the theoretical "hockey stick" figure that characterizes wait times in queue systems, possibly because we stopped our tests with 15 customers, but also because the way we generated random customers does not quite fit the Poisson distribution required by queue theory. It is also interesting to notice the extreme variance in maximum response times

Now assume that before opening the shop we estimate that we will have 3 customers on average every second. How many barbers will we need?



The optimal number of barbers will be about 6, which is perhaps higher than someone who is unfamiliar with queue behaviors may expect – shouldn't 3 barbers who can handle 3 customers a second be able to support 3 concurrent customers with no wait time?
But the most surprising result is the behavior when we allocate too many barbers to the task. At the ridiculous number of 8 barbers with average number of only 3 customers per second, the response times are increasing again. This is an effect of overcoming one major bottleneck (number of barbers in this case), only to run into a number of new bottlenecks (CPU, latches) that cause variance in the system behavior. This problem is known as **thrashing**, where more resources accomplish less work, and this is **Sin 7**. DBAs who hoped to solve a performance problem in a system with IO or latch contention by adding CPUs, only to run into a worse performance problem on a less predictable system, should recognize this graph. This is another reminder that concurrency problems should be detected by looking not only on the behavior of the system on average but also by looking at the extreme values (90th percentile, maximum) and at the variance of the system.

Read/Write Consistency

These are classical concurrency problems that are unique to Oracle, and involve the redo, undo and read consistency mechanisms that Oracle uses. No self-respecting review of concurrency problems in Oracle will be complete without at least mentioning these concurrency issues, but I will only cover these problems very briefly. They are covered in significant depth by Tom Kyte [5].

Sin 8: Non-transactional Changes

Non-transactional changes are perhaps the most severe sin a database developer can commit (pun intended). Transactions are one of these features that make a database what it is. One of the key properties of transactions is atomicity – all the changes done within a transaction are either committed or rolled-back together. Oracle guarantees this behavior automatically, so it is normally one of those things developers don't have to worry about. Except that Oracle does allow you to make changes within a transaction that cannot be rolled back when the transaction is rolled back, leaving the system with "orphan changes". Examples for such non-transactional changes are sending emails, writing to a file, and doing anything within autonomous transaction. This is especially harmful when the non-transactional change is done within a trigger.

Why do I say this is a concurrency problem? Because concurrent changes, can cause rollbacks the developer cannot anticipate in advance, and that will cause triggers to fire twice. If the trigger contains a non-transactional change, the change will occur twice since it was impossible to roll it back on the first run.

Consider the following scenario:
We have a rather large table, with 250,000 rows. Session 1 updates column X on row 250,000. At about the same time, session 2 starts, updating all columns where X has a certain value. When session 2 gets to row 250,000, it will block, waiting for session 1 to commit. Once session 1 commits, session 2 will notice that the row it waited for now contains newer data and that the newer data may change the decision whether or not to update this row. It is impossible to decide whether or not to change this row at this situation, since either decision can lead to inconsistency. The only choice is to roll back session 2 and try again. This retry is called an **update restart**, and is a normal, expected behavior of Oracle.

Of course, if session 2 caused triggers to fire with non-transactional changes during the update, these triggers will fire twice, a slightly less expected behavior.

Sin 9: Unexpected extra IO

High levels of concurrency can cause higher than expected IO in three scenarios:

1. Consistent gets – If there are many updates and selects happening concurrently on the same block, every query will require reads from the undo tablespace to see a consistent image of the data. You should expect to read at least as many extra blocks as there are transactions concurrently modifying our data.

2. The update restarts we just saw.
3. Delayed block cleanout – Suppose a transaction updates blocks in a system that is busy scanning large tables. It is possible that by the time the session commits, the block it changed is no longer in the buffer cache, instead they were written dirty to the disk. In this case, Oracle will only close the transaction in the redo log. Remember, that Oracle keeps lock information in the data block itself, so the data block is written to the disk with the lock information intact. This does not cause problems, as the locking transaction no longer exists. However the next time a query reads this block, Oracle will clean the extra information from the block and write it to disk. This means some extra IO and also extra redo. Note that this effect can cause even select statements to generate redo.

Summary

1. Race conditions only occur when the system is tested with concurrent users. Therefore, the only wait to catch them is by paying attention to unexpected hangs, delays or wrong results during load tests. This could indicate race conditions. If a result happens only under load and is completely irreproducible in a development environment, it is a good indication of a race condition.
2. Race conditions are usually prevented by making the shared resources mutually exclusive. While this solves the race condition, it serializes the code and degrades performance. Sometimes this is an unavoidable tradeoff, but a better solution would be to use a resource that can be shared safely between concurrent processes instead, or two split the resources in advance between the different processes so no sharing will be needed.
3. Deadlock is the best known and researched concurrency problems. It is also the simplest to detect and resolve.
4. Serialized code can be detected by examining traces taken while it was running (you should be able to detect wait events resulting from serialization), or by noticing that the second user takes twice as long to run as the first one.
5. “Dining Philosophers” problem models concurrency at a very scalable system – the more users you have, the more resources they can share. A bit like nodes in RAC that add more memory to the system, which is later shared by all nodes.
6. Starvation, or unfair resource allocation, is more difficult to detect. Large variance in response times for users than ran concurrently can indicate an issue.
7. Since unindexed foreign keys, which lead to over-locking, is the most common reason for deadlocks (and serialization) in Oracle, you can easily prevent most of your problems by indexing correctly.
8. Concurrent systems can be viewed as queue systems. Processes will queue for CPU, locks, latches, IO, etc. We can use queue theory to predict how many users our system can serve at which response times, and also to decide how to size the hardware to serve given number of users.

Scripts

These are the scripts used to generate the results shown in this paper. Note that the code snippets shown in the paper itself were edited for readability and are therefore not exactly identical to the code that appears in the scripts. As you'll see in my comments below, none of it is production quality, but you shouldn't run this on production anyway.
This would be a good time to thank Rob van Wijk for reviewing my coding and suggesting numerous improvements to the code style, it is actually readable now.

- shared_account.sql reproduces the first race condition example. Run it in sqlplus, and it will run shared_account2.sql as the second concurrent session.
- dining_philosophers.sql contains the framework for the dining philosophers problem.
- start_philosophers_1.bat runs the deadlock scenario (a scenario that will cause a deadlock, at least some of the time).
- start_philosophers_2.bat runs the starvation scenario.
- start_philosophers_3.bat runs the effective solution
- All these batch scripts run philosophers_report.sql at the end to generate a nice summary of response times.
- barbershop.sql contains the framework for the barbershop tests.
- Barber_runner.pl runs the scenarios used to demonstrate the different queue situations.

References

[1] http://en.wikipedia.org/wiki/Computer_multitasking
[2] Andrew Tanenbaum(1992). Modern Operating Systems. Engelwood Cliffs, NJ: Prentice Hall.
[3] http://download.oracle.com/docs/cd/B28359_01/server.111/b28318/consist.htm#CNCPT1331
[4] http://jonathanlewis.wordpress.com/2008/05/12/synchronisation/
[5] Tom Kyte (2005). Expert Oracle Database Architecture. Berkeley, CA: Apress.
[6] Mark Bobak, Understanding and Interpreting Deadlocks, HotSos 2006. http://www.oaktable.net/getFile/149
[7] Cary Millsap with Jeff Holt (2003). Optimizing Oracle Performance. Sebastopol, CA: O'Reilly.
[8] Craig Shallahamer (2007). Forecasting Oracle Performance. Berkeley, CA: Apress.